TÉCNICAS DE PROGRAMACIÓN SEGURA

1. Técnicas de seguridad. VISIÓN GENERAL **1**
1.1 Criptografía 1
1.2 Control de acceso 4

2. Seguridad en el entorno JAVA **5**

3. CRIPTOGRAFÍA CON JAVA **9**
3.1. Resúmenes de mensajes **9**
ACTIVIDAD 12
ACTIVIDAD 15

Algunas de las técnicas más importantes a tratar para asegurar sistemas y aplicaciones son:

- La criptografía
- Los certificados digitales y
- El control de acceso

También estudiaremos los ficheros de políticas de seguridad en Java y aprenderemos a usar diferentes librerías de Java para implementar la seguridad: l*ibrerías criptográficas* (**JCA y JCE**), la *extensión de sockets seguros* (**JSEE**) y el *servicio de autenticación y autorización de Java* (**JAAS**).

1. Técnicas de seguridad. VISIÓN GENERAL

Algunas de las técnicas y mecanismos más importantes para asegurar sistemas y aplicaciones son: la *criptografía*, los *certificados digitales* y el *control de acceso*.

1.1 Criptografía

El término criptografía es un derivado de la palabra griega kryptos, que significa “oculto”. El objetivo de la criptografía es ocultar el significado de un mensaje mediante el cifrado o codificación del mensaje.

- Si a un **texto legible** se le aplica un algoritmo de cifrado, que en general depende de una **clave**,

esto arroja como resultado un **texto cifrado** que es el que se envía o guada. A este proceso se le llama **cifrado** o **encriptación**.

• Si a este **texto cifrado** se le aplica el mismo algoritmo , dependiente de la misma **clave** o de otra clave (esto depende del algoritmo) , se obtiene el **texto legible** original. A este segundo proceso se le llama **descifrado** o **desencriptación**.

Existen **3 clases de algoritmos criptográficos**:

• **Funciones de una sola vía** (o funciones *hash*): Estas funciones permiten mantener la integridad de los datos tanto en almacenamiento como en el tráfico de redes. También se utilizan como parte de los mecanismos de **firma digital**. Reciben su nombre debido a su naturaleza matemática. Dado un mensaje x, es muy fácil calcular el resultado de f(x). A este f(x) se le denomina **hash** de x (o traducido podría ser "*resumen*" de x) o también **message digest** de x. La clave está en que es prácticamente imposible calcular x a partir del hash f(x). Las funciones en de una sola vía tienen un amplio abanico de usos en la seguridad informática. Prácticamente cualquier protocolo las usa para procesar claves, encadenar una secuencia de eventos, o incluso autenticar eventos y son esenciales en la **autenticación por firmas digitales**. Los dos algoritmos de una sola vía más utilizados son el **MD5** y **SHA-1**.

• **Algoritmos de clave secreta o de criptografía simétrica:** El emisor y el receptor comparten el conocimiento de una clave que no debe ser revelada a ningún otro. La clase se utiliza tanto para cifrar como para descifrar el mensaje. En estos algoritmos lo esencial es que la clave K, que sirve tanto para cifrar como para descifrar sea compartida solo entre los participantes del sistema. Por eso se dice que la clave es privada o secreta y que los algoritmos son simétricos, pues la misma clave cifra y descifra. El algoritmo de cifrado simétrico más popular es el **DES**, utiliza claves de 56 bits y un cifrado de bloques de 64 bits. Una variante de este es **Triple DES**, la clave es de 128 bits. Otro algoritmo muy utilizado es el **AES** que tiene un tamaño de clave variable siendo el estándar de 256 bits.

• **Algoritmos de clave secreta o de criptografía asimétrica:** El emisor de un mensaje (participante B) emplea una clave pública, difundia previamente por el receptor (participante A), para encriptar el mensaje. El receptor emplea la clave privada correspondiente para desencriptar el mensaje gracias a su clave privada. En general, estos algoritmos se basan en que cada participante genera una **pareja de claves** relacionadas entre sí. Una es la **clave pública** y otra es la **clave privada**, la clave pública todo el mundo la puede conocer, la clave privada solo cada participante conoce su propia clave. Cualquiera que conozca la clave pública del Participante A puede cifrar con ella un mensaje, pero sólo el participante A puede descifrar ese mensaje mediante su clave privada. El algoritmo simétrico más popular es el **RSA** que debe su nombre a sus inventores (*Rivest*, *Shamir* y *Adleman*). Su uso es prácticamente universal como método de autenticación y **firma digital** y es componente de protocolos y sistemas como IPSec, SSL PGP, etc.

Funcionamiento de la firma digital

Una firma digital está compuesta por una serie de datos asociados a un mensaje, estos datos nos permiten asegurar la identidad del firmante (emisor del mensaje) y la integridad del mensaje. El método de firma digital más extendido es el **RSA**.

En este modelo, el procedimiento de firma de un mensaje por parte del emisor es el siguiente:

• El emisor genera un hash(resumen) del mensaje mediante una función acordada, a este hash le llamamos H1.

• Este hash es cifrado con su clave privada. El resultado es lo que se conoce como firma digital (FD) que se envía adjunta al mensaje.

• El emisor envía el mensaje y su FD al receptor, es decir, el mensaje firmado.

El receptor realiza las siguientes operaciones:

• Separa el mensaje de la firma

• Genera el resumen del mensaje recibido usando la misma función que el emisor, se genera H2

• Descifra la firma, FD, mediante la clave pública del emisor obteniendo el hast original, H1.

• Si los dos resúmenes coinciden se puede afirmar que el mensaje ha sido enviado por el propietario de la clave pública utilizada y que no fue modificado en el transcurso de la comunicación.

Todo sistema criptográfico de firma digital descansa sobre un pilar fundamental: la autenticidad de la clave pública de cada participante. Para asegurar la autenticidad de la clave pública de cada participante se recurre a las Autoridades de Certificación (AC), que es la entidad que garantiza que una firma es de quien dice ser.

PUNTOS FUERTES Y PUNTOS DÉBILES.

La siguiente tabla muestra los puntos fuertes y débiles de la criptografía simétrica y asimétrica:

CRIPTOGRAFÍA SIMÉTRICA O DE CLAVE SECRETA

Puntos fuertes

Cifran más rápido que los algoritmos de clave pública.

Sirven habitualmente como base para los sistemas criptográficos basados en hardware.

Puntos débiles

Requieren un sistema de distribución de claves muy seguro (si se conoce la clave se pueden conocer todos los mensajes cifrados con ella) .

En el momento en que la clave cae en manos no autorizadas, todo el sistema deja de funcionar.

Esto obliga a llevar una administración compleja.

Si se asume que es necesaria una clave por cada pareja de usuarios de una red, el número total de claves crece rápidamente con el número de usuarios.

CRIPTOGRAFÍA SIMÉTRICA O DE CLAVE PÚBLICA

Puntos fuertes

Permiten conseguir autenticación y no repudio para muchos protocolos criptográficos.
Suele emplearse en colaboración con cualquiera de los otros métodos criptográficos.
Permiten tener una administración sencilla de claves al no necesitar que haya intercambio de claves seguro.

Puntos débiles

Son algoritmos más lentos que los de clave secreta,con lo que suelen utilizarse para cifrar gran cantidad de datos.
Sus implementaciones son comúnmente hechas en sistemas software.
Para una gran red de usuarios y/o máquinas se requiere un sistema de certificación de la autenticidad de las claves públicas.

En resumen podemos decir que la criptografía juega 3 papeles principales en la implementación de sistemas seguros:

• se emplea para mantener el secreto y la integridad de la información donde quiera que pueda estar expuesta a ataques,

• como base para los mecanismos para autenticar la comunicación entre pares de principales (un principal puede ser un usuario o un proceso) y

• para implementar el mecanismo de firma digital.

1.2 Control de acceso

En general cualquier empresa u organización tiene diversos tipos de recursos de carácter privado y secreto que necesitan asegurar, de manera que solo las personas indicadas puedan acceder a ellos para realizar las tareas requeridas. Estos recursos pueden ser ´físicos (por ejemplo un equipo informático muy caro), informativos (datos confidenciales) o de personal (los empleados).

Se hace necesario realizar un control de acceso a los recursos, pero este control de acceso es algo más que simplemente requerir nombres de usuario y contraseñas. Hay 3 componentes importantes de control de acceso:

- **Identificación:** proceso mediante el cual el sujeto suministra información diciendo quién es.
- **Autentificación:** es cualquier proceso por el cual se verifica que alguien es quien dice ser. Esto implica generalmente un nombre de usuario y una contraseña, pero puede

incluir cualquier otro método para demostrar la identidad, como una tarjeta inteligente, exploración de retina, reconocimiento de voz o huellas dactilares.

- **Autorización:** es el proceso de determinar si el sujeto, una vez autentificado, tiene acceso al recurso. La autorización es equivalente a la comprobación de la lista de invitados a una fiesta.

El sistema de control de acceso debe permitir el acceso al usuario correctamente autentificado y debe impedir el acceso a los demás. Debería también guardar un buen registro de auditoría de todas las entradas e intentos fallidos.

Las medidas de identificación y autentificación se centran en una de estas 3 formas:

- Algo que se sabe, algo que se conoce, típicamente contraseñas, es la más extendida.
- Algo que se es, medidas que utilizan la biometría (identificación por medio de la voz, la retina, la huella dactilar, geometría de la mano, etc).
- Algo que se tiene, los access tokens (sistemas de tarjetas).

2. Seguridad en el entorno JAVA

La máquina virtual Java (Java Virtual Machine) es la encargada de ejecutar un programa Java, esta tarea consiste en interpretar y ejecutar los *bytecodes* (código abierto), es decir, en transformarlos en código de sistema y ejecutar conforme se van interpretando.

Antes de que la JVM comience este proceso de interpretación debe realizar una serie de tareas para preparar el entorno en el que el programa se ejecutará. Este es el punto en el que se implementa la seguridad interna de Java. Hay tres componentes en proceso:

• **El cargador de clases**: Es el responsable de encontrar y cargar los bytecodes que definen las clases. Cada programa Java tiene como mínimo tres cargadores: el cargador de clases bootstrap que carga las clases del sistema (normalmente desde el fichero JAR rt.jar), el cargador de clases de extensión que carga una extensión estándar desde el directorio jre/lib/ext y el cargador de clases de la aplicación que localiza las clases y los ficheros JAR/ZIP de la ruta de acceso a las clases (según está establecido por la variable de entorno CLASSPATH o por la opción -classpath de la línea de comandos.

• **El verificador de ficheros de clases**: Se encarga de validar los bytecodes. Algunas de las comprobaciones que lleva a cabo son: que las variables estén inicializadas antes de ser utilizadas, que las llamadas a un método coinciden con los tipos de referencias de objeto que no se han infringido las reglas para el acceso a los métodos y clases privados, etc.

• **El gestor de seguridad** (o administrador de seguridad). Es una clase que controla si está permitida una determinada operación. Alguna de las operaciones que comprueba son: si el hilo actual puede cargar un subproceso, puede acceder a un paquete específico, puede acceder o modificar las propiedades del sistema, puede leer desde o escribir en un fichero específico, puede eliminar un fichero específico, puede aceptar una conexión socket desde un host y un número de puerto específico, etc. Nos centraremos en este componente.

Por defecto, no se instala de forma automática ningún gestor de seguridad cuando se ejecuta una aplicación Java. En el siguiente ejemplo veremos la salida que produce el programa ejecutándose sin gestor de seguridad y con gestor de seguridad.

***Ejemplo 1*:** El programa muestra los valores de ciertas propiedades del sistema (usamos el método System.getProperty(propiedad) para mostrar los valores), la siguiente tabla describe alguna de las más importantes:

PROPIEDAD	SIGNIFICADO
file.separator	Separador de directorios ("/" en UNIX y "\" en Windows)
java.class.path	Ruta usada para encontrar los directorios y ficheros JAR que cotnienen los archivos de clase
java.home	Directorio para JRE
java.vendor	Nombre del proveedor
java.vendor.url	URL del proveedor
java.version	Número de versión de JRE
line.separator	Fin de línea
os.arch	Arquitectura del sistema operativo
os.name	Nombre del sistema operativo
os.version	Versión del sistema operativo
path.separator	Caracter separador usado en java.class.path (en Unix es: mientras que en windows es;
user.dir	Directorio en el que se está ejecutando el programa Java
user.home	Directorio por defecto del usuario
user.name	Nombre del usuario

La compilación y ejecución muestra la siguiente salida **sin utilizar un gestor de seguridad**:

```
D:\> javac Ejemplo1.java
D:\> java Ejemplo1
```

```
Propiedad: java.class.path
        ==> G:\_INBOX\NetBeansProjects\Ejemplo1\build\classes
Propiedad: java.home
        ==> C:\Program Files\Java\jdk1.8.0_65\jre
Propiedad: java.vendor
        ==> Oracle Corporation
Propiedad: java.version
        ==> 1.8.0_211
Propiedad: os.name
        ==> Windows 10
Propiedad: os.version
  ==> 10.0 Propiedad:
       user.dir
        ==> G:\_INBOX\NetBeansProjects\Ejemplo1
Propiedad: user.home
        ==> C:\Users\usuario
Propiedad: user.name
      ==> usuario
```

Para instalar un gestor de seguridad en nuestro programa podemos incluir al iniciar la JVM la opción **-Djava.security.manager** o invocar al método **setSecurityManager()** de la clase System añadiendo la siguiente línea (en el método main(), por ejemplo antes del bucle for) al código anterior: *System.setSecurityManager(new SecurityMnager()).*

La salida que se genera es la siguiente, donde se pueden observar ciertas propiedades del sistema a las que se ha denegado el acceso.

```
D:\> java -Djava.security.manager ejemplo1.Ejemplo1
Propiedad: java.class.path

          ExcepciÃ3n java.security.AccessControlException: access denied
("java.util.PropertyPermission" "java.class.path" "read") Propiedad:
java.home

          ExcepciÃ3n java.security.AccessControlException: access denied
("java.util.PropertyPermission" "java.home" "read") Propiedad: java.vendor
          ==> Oracle Corporation
Propiedad: java.version
          ==> 1.8.0_65
```

```
Propiedad: os.name
           ==> Windows 10
Propiedad: os.version
  ==> 10.0 Propiedad:
        user.dir

           ExcepciÃ3n java.security.AccessControlException: access denied
("java.util.PropertyPermission" "user.dir" "read") Propiedad: user.home

           ExcepciÃ3n java.security.AccessControlException: access denied
("java.util.PropertyPermission" "user.home" "read") Propiedad: user.name

           ExcepciÃ3n java.security.AccessControlException: access denied
("java.util.PropertyPermission" "user.name" "read")
```

Al ejecutar un programa java se carga por defecto un fichero de políticas predeterminado y otorga todos los permisos al código para acceder a algunas propiedades comunes útiles, tales como os.name y java.version. Estas propiedades no son sensibles a la seguridad, por lo que la concesión de estos permisos normalmente no representa un riesgo de seguridad. Otras propiedades como user.home y java.home, no están entre las propiedades por las cuales el sistema de ficheros de políticas otorga permiso de lectura. Por lo tanto, cuando el programa intenta acceder a ellas, el gestor de seguridad impide el acceso y lanza la excepción **AccessControlException**. Esta excepción indica que la política vigente, que consiste en entradas en uno o más ficheros de políticas, no permite el permiso para leer la propiedad java.home, java.class.path, user.home, etc.

La plataforma Java define un conjunto de APIs que abarca las principales áreas de seguridad, incluyendo la criptografía, infraestructura de clave pública la autenticación, la comunicación segura, y control de acceso. Estas APIs permiten a los desarrolladores integrar fácilmente la seguridad en sus aplicaciones. Algunas de estas APIs son:

- **JCA(Arquitectura criptográfica de Java)**: Es un marco de trabajo para acceder y desarrollar funciones criptográficas en la plataforma Java. Proporciona la infraestructura para la ejecución de los principales servicios de cifrado, incluyendo las firmas digitales, resúmenes de mensajes (hashs), certificados y validación de certificados, encriptación (cifrado de bloques, cifrado simétrico y asimétrico), generación y gestión de claves y generación segura de números aleatorios.
- **JSSE (Extensión de Sockets Seguros Java).** Es un conjunto de paquetes Java provistos para la comunicación segura en Internet. Implementa una versión Java de los protocolos SSL y TSL, además incluye funcionalidades como cifrado de datos, autenticación del servidor, integridad de mensajes y autenticación del cliente.
- **JAAS (Servicio de Autentificación y Autorización de Java)**: Es una interfaz que permite a las

aplicaciones Java acceder a servicios de control de autenticación y acceso. Puede usarse con dos fines: la autenticación de usuarios para conocer quién está ejecutando código Java; y la autorización de usuarios para garantizar que quién lo ejecuta tiene los permisos necesarios para hacerlo.

3. CRIPTOGRAFÍA CON JAVA

El API **JCA** (incluye la extensión criptográfica de Java **JCE**) incluida dentro del paquete JDK incluye dos componentes software:

- El marco que define y apoya los servicios criptográficos para que los prestadores faciliten implementaciones. Este marco incluye paquetes como **java.security**, **javax.crypto**, **javax.crypto.spec** y **java.crypto.interfaces**.
- Los proveedores reales, tales como Sun, SunRSASign, SunJCE, que contienen las implementaciones criptográficas reales. El proveedor es el encargado de proporcionar la implementación de uno o varios algoritmos al programador. Los proveedores de seguridad se definen en el fichero **java.security** localizado en la carpeta **java.home\lib\security**\: forman una lista de entradas con un número que indican el orden de búsqueda cuando en los programas no se especifica un proveedor.;

```
security.provider.1 = sun.security.provider.Sun
security.provider.2 = sun.security.provider.rsa.SunRsaSign
security.provider.3 = com.sun.net.ssl.internal.ssl.Provider
security.provider.4 = com.sun.crypto.provider.SunJCE
security.provider.5 = sun.security.jgss.SunProvider
```

JCA define el concepto de proveedor mediante la clase Provider de paquete java.security. Se trata de una clase abstracta que debe ser redefinida por las clases proveedor específicas. Tiene métodos para acceder al nombre del proveedor, número de versión y otras informaciones sobre las implementaciones de los algoritmos, para la generación, conversión y gestión de claves y la generación de firmas y resúmenes.

3.1. Resúmenes de mensajes

Un message digest o resumen de mensaje (también se le conoce como función hash) es una marca digital de un bloque de datos. Existe un gran número de algoritmos diseñados para procesar estos message digests, los dos más conocidos son **SHA1** y **MD5**.

La clase **MessageDigest** permite a las aplicaciones implementar algoritmos de resumen de mensajes, como MD5, SHA-1 o SHA-256. Dispone de un constructor protegido, por lo que se

accede a él mediante el método *getInstance(String algoritmo)*. Algunos de sus métodos se exponen en la siguiente tabla:

MÉTODOS	**MISIÓN**
public static MessageDigestgetInstance (String algoritmo) **public static MessageDigestgetInstance (String algoritmo, String proveedor)**	Devuelve un objeto *MessageDigest* que implementa el algoritmo de resumen especificado. En el primer caso, los proveedores de seguridad se buscan según el orden establecido en el fichero **java.security**. En el segundo caso se busca el proveedor dado. Nombres válidos para el proveedor de seguridad predeterminado de Sun son SHA, SHA-1 y MD5. Puede lanzar la excepción *NoSuchAlgorithmException* si no hay proveedor que implemente el algoritmo dado. Si el nombre de proveedor no se encuentra se produce *NoSuchProviderException.*
void update(byte input)	Realiza el resumen del byte especificado.
void update(byte [] input)	Realiza el resumen del array de bytes especificado.
byte[] digest	Completa el cálculo del valor hast, devuelve el resumen obtenido.
byte [] digest (byte [] entrada)	Realiza una actualización final sobre el resumen utilizando el array de bytes indicado en el argumento, y luego completa el cálculo de resumen .
void reset()	Reinicializa el objeto resumen para un nuevo uso .
int getDigestLength()	Devuelve la longitud del resumen en bytes, o 0 si la operación no está soportada por el

	proveedor.
String getAlgorithm()	Devuelve un String que identifica el algoritmo
Provider getProvider()	Devuelve el proveedor del objeto.
static boolean isEqual(byte[] digesta, byte digestb)	Comprueba si dos mensajes resumen son iguales. Devuelve true si son iguales y false en caso contrario .

El ejemplo que se muestra a continuación genera el resumen de un texto plano. Con el método MessageDigest.getInstance("SHA")se obtiene una instancia del algoritmo SHA con el que se hará el resumen. El texto plano lo pasamos a un array de bytes y el array se pasa como argumento al método update(), finalmente aplicando el método digest() se obtiene el resumen del mensaje. Después se muestra en pantalla el número de bytes generados en el mensaje, el algoritmo utilizado, el resumen generado y convertido a hexadecimal mediante el método hexadecimal() y por último información del proveedor.

```
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import java.security.Provider;
/** * * @author usuario */ public class Ejemplo4 {

    /**
    * @param args the command line arguments */
    public static void main(String[] args) {
        // TODO code application logic here
        MessageDigest md;
        try{
            md=MessageDigest.getInstance("SHA");
            String texto="Esto es un texto plano.";
            byte dataBytes[]= texto.getBytes(); // texto a abtes
            md.update(dataBytes); // se introduce texto en bytes a resumir byte
            resumen[]= md.digest(); // se calcula el resumen

        System.out.println("Mensaje original: " + texto);

        System.out.println("Número de bytes " + md.getDigestLength());
        System.out.println("Algoritmo: " + md.getAlgorithm());

        System.out.println("Mensaje resumen: " + new String(resumen));
        System.out.println("Mensaje en hexadecimal: " + Hexadecimal(resumen));

        Provider proveedor = md.getProvider();

        System.out.println("Proveedor : " + proveedor.toString()); }
```

```
    catch(NoSuchAlgorithmException e) {
          e.printStackTrace();
      }

 }// main


 // convierte un array de bytes a hexadecimal
 static String Hexadecimal(byte[] resumen) {
     String hex=""; for (int i = 0; i < resumen.length; i++) {
         String h=Integer.toHexString(resumen[i] & 0xFF);
         if (h.length()==1) hex+="0"; hex+=h;
         }
         return hex.toUpperCase();
     }// hexadecimal
}
```

La ejecución muestra la siguiente salida:

```
Mensaje original: Esto es un texto plano.
Número de bytes 20
Algoritmo: SHA Mensaje resumen: q # _ j K~ ?? ??? ? ????? ?
Mensaje en hexadecimal: 71878D1BBEB2D65FE1886AF4DDEE85D2D74B7EC7
Proveedor : SUN version 1.8
```

Se puede crear un resumen cifrado con clave usando el segundo método digest(bytes[]) donde se proporciona la clave en un array de bytes, en este caso el mensaje resumen generado será diferente al generado anteriormente.

```
String clave=”clave de cifrado”;
byte dataBytes[] = texto.getBytes();
md.update(dataBytes); // Se introduce texto en bytes a resumir byte resumen [] =
md.digest(clave.getBytes());
```

ACTIVIDAD

Prueba el ejercicio anterior con el algoritmo MD5. Comprueba el número de bytes generados.

Supongamos que ahora queremos guardar un objeto String en un fichero, pero queremos

estar seguros de que a la hora de leer el String el fichero no esté dañado o no haya sido manipulado y los datos sean correctos.

EJEMPLO 5.

```
package ejemplo5;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectOutputStream;
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;

public class Ejemplo5 {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) throws NoSuchAlgorithmException {
        try {
            FileOutputStream fileout=new FileOutputStream("DATOS.DAT");
            ObjectOutputStream dataOS=new ObjectOutputStream(fileout);
            MessageDigest md=MessageDigest.getInstance("SHA");
            String datos="En un lugar de la Mancha, " +
                    "de cuyo nombre no quiero acordarme, no ha mucho tiempo "  +
                    "que vivia un hidalgo de los de lanza en astillero, "
                    + "adarga antigua, rocín flaco y galgo corredor.";
            byte dataBytes[]=datos.getBytes();
            md.update(dataBytes); // Texto a resumir
            byte resumen[]=md.digest(); // se calcula el resumen
            dataOS.writeObject(datos); // se escriben los datos
            dataOS.writeObject(resumen); // se escribe el resumen
            dataOS.close();
            fileout.close();
        }
        catch (IOException e) {
            e.printStackTrace();
        }
        catch(NoSuchAlgorithmException e) {
            e.printStackTrace();
        }
    }

}
```

EJEMPLO 6

Al recuperar los datos del fichero primero necesitaremos leer el String y luego el resumen, a continuación hemos de calcular el resumen con el String leido y comparar este resumen con el leído del fichero.

```
package ejemplo6;
import java.io.FileInputStream;
import java.io.ObjectInputStream;
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
public class Ejemplo6 {
        public static void main(String[] args) throws NoSuchAlgorithmException {
```

```
            try {
              FileInputStream fileout = new FileInputStream("DATOS.DAT");
              ObjectInputStream dataOS=new ObjectInputStream(fileout);
              Object o = dataOS.readObject();
              // primera lectura, se obtiene el String
              String datos = (String) o;
              System.out.println("Datos: " + datos);
              // Segunda lectura, se obtiene el resumen
              o=dataOS.readObject();
              byte resumenOriginal[]=(byte[]) o;

              MessageDigest md=MessageDigest.getInstance("SHA");
              // Se calcula el resumen del String leido del fichero
              md.update(datos.getBytes()); // texto a resumir
              byte resumenActual[]=md.digest(); // se calcula el resumen
              // se comprueban los dos resúmenes
              if (MessageDigest.isEqual(resumenActual, resumenOriginal))
                  System.out.println("DATOS VÁLIDOS");
              else
                  System.out.println("DATOS NO VÁLIDOS");
              dataOS.close();
              fileout.close();
            } catch (Exception e ) {
              e.printStackTrace();
            }
        }

}
```

ACTIVIDAD

Ejecuta el Ejemplo 5 para generar el fichero DATOS.DAT. Después ejecuta el Ejemplo 6. Edita el fichero DATOS.DAT y cambia alguna letra, por ejemplo donde hay una minúscula pon mayúscula y prueba de nuevo el Ejemplo 6 para ver su salida.

3.2. Generando y verificando Firmas Digitales

El resumen de un mensaje no nos da un alto nivel de seguridad. En el ejemplo anterior podemos decir que el fichero no es correcto si el texto que se lee no produce la misma salida que el resumen guardado. Pero alguien puede cambiar el texto y el resumen, y no podemos estar seguros de que el texto sea el que debería ser.

En este apartado veremos cómo las firmas digitales pueden autentificar un mensaje y asegurar que el mensaje no ha sido alterado y procede del emisor correcto. Para crear una firma digital se necesita una firma privada y la clave pública correspondiente con el fin de verificar la autenticidad de la firma.

En algunos casos, el par de claves (pública y privada correspondiente) están disponibles en ficheros. En ese caso, el programa puede importar y utilizar la clave privada para firmar. En otros casos, el programa necesita generar el par de claves. La clase **KeyPairGenerator** nos permite generar el par de claves.

La clase **KeyPair** es una clase soporte para generar claves públicas y privada. Dispone de los métodos **PrivateKey getPrivate()** y **PublicKey getPublic()**.
PrivateKey y **PublicKey** son interfaces que agrupan todas las interfaces de clave privada y pública respectivamente.

Para firmar los datos se usa la clase **Signature**. Un objeto de esta clase se puede utilizar para generar y verificar firmas digitales.

Al especificar el nombre del algoritmos de firma, también se debe incluir el nombre del algoritmo de resumen de mensajes utilizado por el algoritmo de firma. *SHA1withDSA* es una forma de especificar el algoritmo de firma DSA, usando el algoritmo de resumen SHA-1.

Existen 3 fases en el uso de un objeto Signature ya sea para firmar o verificar datos: inicialización (ya sea con clave pública *initVerify()*) o clave privada *initSign()*), actualización(*update()* y firma *sign()*) o verificación (*verify()*).

EJEMPLO 7

```
import java.security.InvalidKeyException;
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.NoSuchAlgorithmException;
import java.security.PrivateKey;
import java.security.PublicKey;
import java.security.SecureRandom;
import java.security.Signature;
import java.security.SignatureException;
import java.util.logging.Level;
import java.util.logging.Logger;
public class Ejemplo7 {
    public static void main(String[] args) throws NoSuchAlgorithmException {
        try {

            //OBTENEMOS EL OBJETO GENERADOR
            KeyPairGenerator KeyGen = KeyPairGenerator.getInstance("DSA");
            // SE INICIALIZA EL GENERADOR DE CLAVES
            SecureRandom numero = SecureRandom.getInstance("SHA1PRNG");
            //EL TAMAÑO DEBE ESTAR ENYTRE 512 Y 1024 Y DEBE SER MÚLTIPLO DE 64
            //SI SE PASA UN NUMERO INVALIDO DARÁ EXCEPCIÓN INVALIDPARAMETEREXCEPTION
            KeyGen.initialize(1024, numero);
            // SE CREA EL PAR DE CLAVES PRIVADA Y PÚBLICA
            KeyPair par = KeyGen.generateKeyPair();
            PrivateKey clavepriv = par.getPrivate();
            PublicKey clavepub = par.getPublic();
            //FIRMA CON CLAVE PRIVADA EL MENSAJE
            // AL OBJETO Signature SE LE SUMINISTRAN LOS DATOS A FIRMAR
            Signature dsa = Signature.getInstance("SHA1withDSA");
            dsa.initSign(clavepriv);
            String mensaje = "Este mensaje va a ser firmado";
            dsa.update(mensaje.getBytes());
```

```
            byte[] firma = dsa.sign(); //MENSAJE FIRMADO
            // EL RECEPTOR DEL MENSAJE
            // VERIFICA CON LA CLAVE PÚBLICA EL MENSAJE FIRMADO
            // AL OBJETO Signature SE LES SUMINIST. LOS DATOS A VERIFICAR

            Signature verificadsa = Signature.getInstance("SHA1withDSA");
            verificadsa.initVerify(clavepub);
            verificadsa.update(mensaje.getBytes());
            boolean check = verificadsa.verify(firma);
            if (check)
               System.out.println("FIRMA VERIFICADA CON CLAVE pÜBLICA");
            else
                System.out.println("FIRMA NO VERIFICADA");
           }
           catch (NoSuchAlgorithmException e1) {
              e1.printStackTrace();
           } catch (InvalidKeyException e) {
            e.printStackTrace();
           } catch (SignatureException e) {
            e.printStackTrace();
        }
    }//main
} // Ejemplo 7
```

ALMACENAR LAS CLAVES PÚBLICA Y PRIVADA EN FICHEROS

Para almacenar la clave privada en disco, en el siguiente ejemplo guarda la clave privada en un fichero en disco de nombre Clave.privada:

```
PKCS8EncodedKeySpec clavePrivadaSpec = new PKCS8EncodedKeySpec(clavepriv.getEncoded());
//Escribir a fichero binario la clave privada
FileOutputStream outpriv = new FileOutputStream("Clave.privada");
outpriv.write(clavePrivadaSpec.getEncoded());
outpriv.close();
```

Para almacenar la clave pública en disco es necesario codificarla en formato X.509, en el siguiente ejemplo guarda la clave pública en un fichero en disco de nombre Clave.publica:

```
X509EncodedKeySpec clavePublicaSpec = new X509EncodedKeySpec(clavepub.getEncoded());
//Escribir a fichero binario la clave pública
FileOutputStream outpub = new FileOutputStream("Clave.publica");
outpub.write(clavePublicaSpec.getEncoded());
outpub.close();
```

ACTIVIDAD

Almacena las clases públicas y privadas en disco en los ficheros Clave.publica y Clave.privada respectivamente.

FIRMAR LOS DATOS DE UN FICHERO CON LA CLAVE PRIVADA: EJEMPLO 8.

La firma se almacenará en un fichero FICHERO.FIRMA.

VERIFICAR LA FIRMA DE UN FICHERO CON LA CLAVE PÚBLICA: EJEMPLO 9.

Necesitaremos la clave pública almacenada en el fichero Clave.publica, la firma del fichero almacenada en FICHERO.FIRMA y el fichero de datos FICHERO.DAT. En el primer lugar obtendremos la clave pública del fichero Clave.publica, a continuación obetndremos la firma digital almacenada en el fichero FICHERO.FIRMA. A continuación se leen los datos del fichero de datos FICHERO.DAT y se suministran al objeto Signature. Por último se verifica la firma con la clave pública.